A System for Assisting Program Transformation

MARTIN S. FEATHER
University of Edinburgh

Program transformation has been advocated as a potentially appropriate methodology for program development. The ability to transform large programs is crucial to the practicality of such an approach.

This paper describes research directed toward applying one particular transformation method to problems of increasing scale. The method adopted is that developed by Burstall and Darlington, and familiarity with their work is assumed.

The problems which arise when attempting transformation of larger scale programs are discussed, and an approach to overcoming them is presented. Parts of the approach have been embodied in a machine-based system which assists a user in transforming his programs. The approach, and the use of this system, are illustrated by presenting portions of the transformation of a compiler for a "toy" language.

Categories and Subject Descriptors: D.1.1. [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; D.3.4. [**Programming Languages**]: Processors—*optimization*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Design, Languages

Additional Key Words and Phrases: program transformation, program development

1. INTRODUCTION

Motivated by the increasingly significant cost of software development and maintenance in the computing environment, researchers have sought methodologies for program development. One such methodology that has been advocated is that of program transformation (see, e.g., [1, 2, 10]). The application of program transformation is not limited to assisting software development. Other areas include

(1) investigating classes of algorithms (e.g., synthesis of several sorting algorithms from a single specification [8], list-copying algorithms [19]);
(2) assisting program description and verification (e.g., proof of a list-copying algorithm by developing it through transformation and proving the correctness of each transformation step [21]);
(3) adapting existing programs (e.g., adaption and maintenance of mathematical software packages [4]).

The British Science Research Council supported this research and provided computing facilities.
Author's address: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291.

Our primary interest within this paper is directed only toward the application of transformation to assist software development.

We adopt one particular transformation method, that developed by Burstall and Darlington as presented in [6], and investigate its application to larger sized examples. To keep this paper concise, we assume that the reader is familiar with their work.

We have concentrated upon developing techniques for transforming nontrivial programs and have embodied many of them in our transformation system ZAP. As evidence that this has enabled us to tackle sizable transformations, we cite our success in using ZAP to transform an entire text-formatting program.

In Section 2 we consider the implications of adopting Burstall and Darlington's method as a basis for software development. In Section 3 we raise the issues relevant to increasing the scale of programs to be transformed. Our approach and system are described in Section 4; portions of our transformation of a compiler for a "toy" language illustrate this. In Section 5 we review the approach and its successes, limitations, and possible extensions.

2. IMPLICATIONS OF ADOPTING BURSTALL AND DARLINGTON'S TRANSFORMATION METHOD

Burstall and Darlington's method consists of the repeated application of a small number of simple manipulations to recursion equations to produce modified recursion equations.

Two immediate consequences of adopting their technique are that the program to be transformed must be written in recursion equations and that, after transformation, we are left with a program still in recursion equations (in particular, without side-effects, assignment, etc.). The former implies a restriction on the nature of the programs we may propose for transformation; the latter implies that the end result of our transformations will typically require further processing (outside the scope of this method) in order to obtain a program in some conventional imperative language making use of iteration and destructive data operations.

We point out that there is nevertheless a wide difference between the structure of recursive programs proposed as specifications and the structure of the transformed programs we emerge with. Our approach is to write a (very) straightforward algorithm to solve the problem in recursion equations, test this algorithm on small examples, and, having satisfied ourselves that it exhibits the behavior we desire, transform it with the intention of attaining a (typically grossly changed) structure close to that of an efficient imperative solution.

The recursion equations formed the basis of a language, NPL, for which Burstall implemented an interpreter ([5]). The spirit of NPL has since been incorporated into a new language, HOPE, described in [7]. Our experience with writing straightforward recursion equation programs has been encouraging; once we have the programs syntactically correct, we find that they are usually semantically correct.

Conversion of a purely applicative recursion equation program into an imperative program is necessary to achieve two effects: first, to replace recursion with iteration, and second, to make use of destructive operations to overwrite shared

data structures. Certainly, the task of performing such conversion is nontrivial, hence fundamental to the viability of this particular approach. This falls outside the scope of the work we have done. We might hope to achieve the first effect by developing more sophisticated compilers (e.g., some LISP compilers are able to convert tail recursion into iteration). Achieving the second effect is much harder; we might apply different transformation techniques (e.g., [11]) or annotate recursion equations with (mechanically verifiable) assertions which would be sufficient to permit compile-time garbage collection (e.g., [23, 25]).

3. TACKLING LARGER SCALE TRANSFORMATIONS

Whatever transformation method we adopt, if our hope is to use transformation for software development, we must face the issues which arise when we attempt to tackle large-scale problems, including these:

(1) How rapidly does the effort required to transform grow with the scale of the problem, and what can we do to minimize the impact of such growth?

(2) How confident are we in the correctness of our adopted transformation method and the correctness of our application of it?

(3) Does the use of transformation help the process of software maintenance; that is, if we make modifications to our initial program, must we reperform the entire transformation process from scratch, or may we save some unnecessary effort by reusing portions of the first transformation?

We consider the first two issues in the context of adopting Burstall and Darlington's transformation method. The third issue is clearly of great practical significance; however, we have not investigated very far in that direction.

The adopted transformation method consists of the repeated application of small manipulations; hence, in increasing the scale of the program to be transformed, we must necessarily perform a longer sequence of such manipulations. It is readily apparent that we must find a better structure for our transformations than merely long linear sequences. The approach we have developed has been to structure the transformation hierarchically, each layer of the hierarchy expanding into the next layer down, with the bottom layer being the linear sequence of manipulations. At the highest layers we do this expansion by hand. Our transformation system, ZAP, takes over at an intermediate layer and completes the expansion down to manipulation sequences. Thus there are two inputs to ZAP: the simple program serving as specification and the sequence of commands to direct its transformation. We call the sequence of commands a "metaprogram" (since it tells the system how to transform a program). The output is the transformed program; see Figure 1. We wish to stress the existence of metaprograms as manipulable objects. A metaprogram may serve as documentation of how efficiency has been introduced into the transformed program. Retaining a metaprogram allows us to repeat the transformation; modifying a metaprogram allows us to modify the implementation without altering its functional behavior. In the case of a modification to the specification, we may be able to reuse the same metaprogram or see how to adjust it to transform the adjusted specification. The emphasis in the design of ZAP has been to use metaprograms to improve communication between user and system, so that the user can be in control of

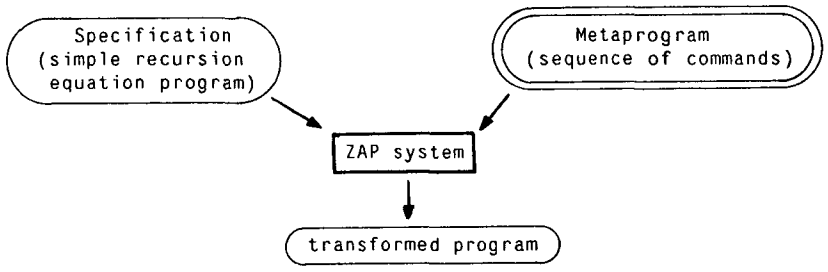


Fig. 1. Input and output of ZAP.

the transformation but can say what he has in mind in a suitably pithy way. In this respect we have taken a very different direction from the more automatic and heuristic approach embodied in Darlington's system [9] and in Manna and Waldinger's DEDALUS system [20].

Provided our hierarchical approach is only permitted to make changes to programs by descending to the level of applying manipulations, we may limit our concern to the correctness of our implementation of these manipulations and to their theoretical correctness. It is clear that Burstall and Darlington's manipulations preserve partial correctness; total correctness is not assured—we may lose termination. In practice, this does not appear to be a danger. Should we wish to be certain, we might seek to prove termination formally. Alternatively, we might try to show that our particular sequence of manipulations is guaranteed to preserve termination. Kott [18] has investigated this latter approach.

4. APPROACH TO TRANSFORMATION

In this section we describe our approach to transformation. The features of our approach are illustrated by means of small examples, and where appropriate by presenting their application to performing transformation of a sizable program. Before considering transformation, we introduce the problem serving as the sizable example and briefly describe our construction of a simple recursion-equation program to serve as the specification.

4.1 The Example Problem

Our task is to produce part of a compiler for a "toy" language: the part to convert abstract syntax trees into machine code. We choose a very simple language with the following constructs:

Statements:	assignment;
	while-loop;
	if–then–else;
	block (headed by local variable declarations).
Expressions:	variable;
	operator applied to a list of expressions.

Our task is very easy; we do not even have procedures or functions within our language. Nevertheless, we are faced with a representative class of compiling problems: evaluating expressions, coding loops and conditionals, and identifying the correct incarnation of local variables defined in blocks.

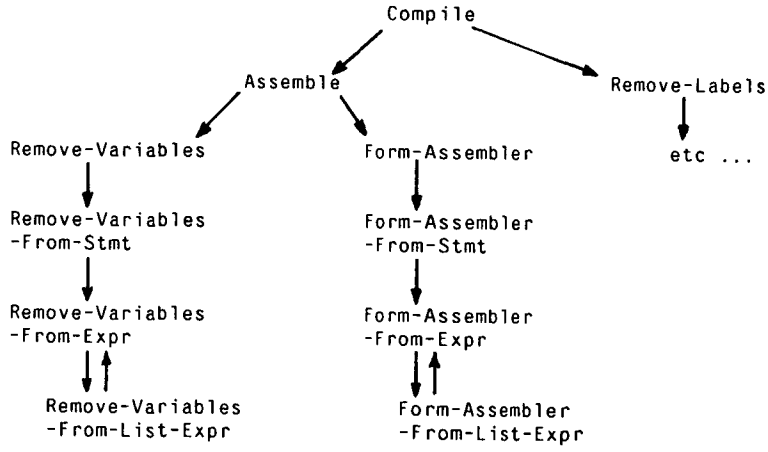


Fig. 2. Structure of initial program.

We assume a stack machine with the following instructions:

load	to load a value from an address onto the stack;
store	to pop the value from the stack and store it in an address;
jump	to jump to an address;
conditional jump	to pop the value from the stack and jump only if that value represents "true";
apply	to apply an operator, which causes enough values to be popped off the stack and the answer to be pushed on;
nonop	to do nothing.

4.2 The Initial Program

Our philosophy for design of the initial program is to split the task into simpler subtasks which communicate with each other in a straightforward manner. Each subtask is further divided until we reach trivial operations for which we can confidently write functions to carry them out. For our compilation problem we are led to an initial program with the structure shown in Figure 2. The overall task is split into two subtasks: first, form assembly code (which will be akin to machine code but for the use of labels instead of explicit locations) from the incoming source statement, and second, replace jumps to labels with jumps to locations in order to obtain machine code from our intermediate assembly code. The first subtask breaks down further into replacing variables within the incoming source statements by locations and then from this forming assembly code to perform each statement's action. The breakdown of the problem continues in this fashion.

We can see that the outcome will be a multipass program, where each pass does some simple activity and communication between the passes is by the handing over of a bulky but conceptually simple data structure. Thus our specification structure models a conceptual breakdown of the task, far removed from the structure an efficient solution to the problem would exhibit. Our transformation techniques must find the path between the two.

4.3 Transformation

Having described the problem, we may now consider how to perform the transformation. When faced with a program of the scale of this compiler, we must plan our approach to its transformation. As outlined earlier, we take a hierarchical approach to structuring the transformation activity. We call the highest level of our hierarchical organization the "strategy" level.

4.3.1 *Transformation Strategies.* Much of the benefit of following a strategy comes from having organized the overall transformation process in a comprehensible manner: during its execution we are better able to gauge our progress, keep track of our objectives, and draw parallels between similar portions of the transformation.

The transformation strategy we have applied to the compiler problem and other examples is a "bottom-up" strategy based on the program structure; we first transform the lowest level functions, then transform the functions that make use of them, and so on. We have not had sufficient experience with other strategies to suggest that this bottom-up strategy is in general better (or even as good as) any other. This strategy has proven reasonably robust, insofar as on all but one occasion (during the transformation of the largest example we have tackled) we have been able to follow it through the entire transformation.

With respect to the compiler example, we concentrate our attention on only a portion of the entire transformation plan: that part dealing with Assemble. The strategy suggests that we first improve the functions it makes use of before tackling it itself. Similarly, we would separately transform Remove-Labels and then be in a position to tackle Compile, which makes use of both of these.

The consideration of transformations to improve individual functions takes us down to the next level in our hierarchy, that of transformation "tactics." The transformation system as implemented does not provide any support for production of strategies or their expansion into tactics. Despite the lack of such support, we have nevertheless found it beneficial to work downward all the way from this strategy level, expending manual effort to do so.

4.3.2 *Transformation Tactics.* At this level the task is to transform an individual function. The scope of the problem has been narrowed to the structure of that function and its use of other functions. We need not consider the context in which it is used, nor (ideally) the entire structure of the functions it uses.

We have found three tactics to be of use frequently; we call them "combining," "tupling," and "generalizing." They are as follows:

The Combining Tactic. This tactic is applicable when the body of the function under consideration contains nested function calls. Its action is first to replace these nested calls with a call to a single (possibly new) function, defined as the nested combination, and then to transform the new function to obtain an immediately recursive definition. The following trivial example serves to illustrate this tactic.

Example. Suppose we are concerned with a function whose definition contains

```
Sum(Squares(NUMLIST))
```

where Sum and Squares are defined by

```
Sum(nil) <= 0
Sum(cons(N,NUMLIST)) <= N + Sum(NUMLIST)

Squares(nil) <= nil
Squares(cons(N,NUMLIST)) <= cons(N*N,Squares(NUMLIST))
```

Then the combining tactic suggests we define a new function:

```
SumSquares(L) <= Sum(Squares(L))
```

Using this, we replace the nested call in the original expression by

SumSquares(NUMLIST)

and then transform SumSquares to obtain

```
SumSquares(nil) <= 0
SumSquares(cons(N,L)) <= N*N + SumSquares(L)
```

The efficiency improvements result from having replaced two function calls by a single call and having completely eliminated the construction and subsequent consumption of the data structure intermediate to the nested function calls.

In our compiler example, this tactic would be applicable when seeking to transform Assemble, since the definition of Assemble is

```
Assemble(STMT,ADR)
          <= Form-Assembler(Remove-Variables(STMT,ADR))
```

In this case there is no need to define a new function, since the body of Assemble *is* the nested function call. We would transform Assemble to derive a recursive version not making use of Form-Assembler or Remove-Variables.

In general, we may have many nested function calls, for example,

$$Fun_1(Fun_2(\ldots Fun_n(\ldots)\cdots)),$$

in which case we might attempt to combine them all at once or, less ambitiously, combine them incrementally.

The Tupling Tactic. This tactic is applicable when the body of the function under consideration contains separate (nonnested) calls to two or more functions, with the calls sharing argument(s). Its action is first to replace these separate calls with a single call to a (possibly new) function defined to return as a result a tuple of the results of the separate calls, and then to transform the new function to obtain an immediately recursive definition. Pettorossi [24] examines this tactic and its implications for efficiency improvement in some detail. This tactic is used during transformation of our compiler program, although not in the portion we examine.

The Generalizing Tactic. This tactic may be applied on any occasion where there is a function call. Its action is to replace the call with a call to a (possibly new) more general function. The benefit of applying this tactic is realized when the original function does not admit to improvement but the more general

function does. Reference [6, app. 1] presents a good example of this. We see a trivial application of this tactic during the compiler program's transformation in the next section.

We draw an analogy with theorem proving, where the strengthening of an induction hypothesis is necessary to permit completion of an inductive proof. Not surprisingly, it is much harder to decide when (and how!) to apply the generalization tactic than the other tactics.

The above tactics have all been recognized as techniques for use in program transformation by other researchers:

(1) combining (also called "functional composition"): [6, 22];
(2) tupling (also "pairing," "functional combination"): [3, 6, 15];
(3) generalizing (also "embedding"): [3, 6, 27].

Consideration of implementing the tactics takes us down to the next level in our hierarchy, that of "pattern-directed" transformations. It is from this point that the ZAP system takes over the work: pattern-directed transformations are expressed as commands to the system, to be expanded into the appropriate manipulations. In practice, it is the task of realizing tactics with pattern-directed transformations that consumes much of the user's time and effort in a transformation; hence, any further development of the system would best be concentrated here.

4.3.3 *Pattern-Directed Transformations.* At this level the task is to carry out the actions suggested by a tactic. The ultimate objective (which we are nearing) is to implement them as linear sequences of manipulations. This level serves as an intermediary between tactics and manipulations.

The two main features of this level are

(1) a mechanism to limit the context of the transformation and
(2) a mechanism to express the objective of the transformation.

These are fully supported by commands to the transformation system. We present these by giving a small example, followed by a look at their application in the transformation of the compiler. The expansion process down to manipulation sequences is discussed in Section 4.3.4.

Returning to the Sum(Squares(NUMLIST)) example, we may be dealing with some function (Foo, say) of the form

Foo(NUMLIST) <= ... Sum(Squares(NUMLIST)) ....

We want to create a new function to be used in place of the nested call. The commands we would issue to our system to do this are as follows:

```
CONTEXT
   UNFOLD  Foo
   USING RESTRICTED  Sum Squares
   TRANSFORM
      GOAL  Foo(NUMLIST) <= $$(&&SumSquares(NUMLIST))
   END
END
```

CONTEXT . . . END establishes the context in which the transformation takes place.

UNFOLD Foo states that the equation for Foo is to be used for unfolding. Occasionally, we wish to use for unfolding the equations for a function and all the functions that are made use of, directly or indirectly, by that function. To achieve this we would use the UNFOLDALL command rather than UNFOLD. Thus, UNFOLDALL Foo would state that the equations of Foo, Sum, SumSquares, *, and + be used for unfolding.

USING RESTRICTED Sum Squares states that functions Sum and Squares may occur in any new function's definition.

TRANSFORM . . . END contains transformation(s) to be performed within the established context. Each transformation is expressed as a GOAL, consisting of two portions separated by "<="; the left-hand side is the expression to be transformed, and the right-hand side, which we call a "pattern," is an approximation of the answer sought. The strange symbols "$$" and "&&" permit us to write approximations. "&&" prefixes the name we wish to give to our new function, and its occurrence marks the position where we want a call to that new function. "$$" marks an arbitrary portion of the expression, with some restrictions; in particular, it may not stand in place of any functions declared as USING RESTRICTED.

The effects of these commands are to cause the new function SumSquares to be defined (and its type declaration made) and Foo's definition to be modified to make use of it.

Next, we want to transform SumSquares itself; this we do as follows:

```
CONTEXT
   UNFOLD  SumSquares Sum Squares
   USING  SumSquares
   TRANSFORM
      GOAL  SumSquares(nil) <= 0
      GOAL  SumSquares(cons(N,L)) <= $$(N,SumSquares(L))
   END
END
```

This states that within the defined context we unfold definitions of SumSquares, Sum, and Squares. USING declares which of the functions being used for unfolding (in this case, SumSquares) may occur in the transformed expression. (The implications of USING and USING RESTRICTED may appear arbitrary. Our only defense is that in practice they provide the desired effects.)

The TRANSFORM . . . END block contains two GOALs, corresponding to the cases into which we wish to split the definition of SumSquares. The first, GOAL SumSquares(nil) <= 0, is the "base case" for our recursive definition of SumSquares. In situations where the pattern is a trivial expression (e.g., a constant, as here) there is no need to provide it; the goal could have been expressed simply as GOAL SumSquares(nil). The second, GOAL SumSquares(cons(N, L)) <= $$(N, SumSquares(L)), corresponds to the recursive case for SumSquares. The provided pattern expresses our wish for an answer somehow involving N and a recursive call SumSquares(L).

Now we consider how to apply these mechanisms to a portion of the compiler

transformation, the transformation of Assemble. At this point we must introduce some data definitions and function definitions.

The data definitions of objects to represent abstract syntax and machine code are given in Figure 3.

There exist analogous definitions for *source-statement'* and *expression'*, which differ only in having explicit addresses in place of variables.

The relevant function definitions are

```
Assemble : source-statement x address -> list assembler
Assemble(STMT,ADR) <= Form-Assembler(Remove-Variables(STMT,ADR))
```

```
Remove-Variables : source-statement x address -> source-statement'
Remove-Variables(STMT,ADR)
                  <= Remove-Variables-From-Stmt(STMT,Empty-Env(ADR))
```

```
Form-Assembler : source-statement' -> list assembler
Form-Assembler(STMT') <= Form-Assembler-From-Stmt(STMT',InitialLabel)
```

In the above, the second parameter of Remove-Variables-From-Stmt is an environment: a data structure containing a mapping from variable names to addresses and the address in store (ADR) from which to allocate space for new variables declared in source statement STMT; Empty-Env(ADR) constructs an initially empty environment. The second parameter of Form-Assembler-From-Statement is a label from which unique new labels are constructed when required.

The transformation we wish to perform is to improve Assemble by combining Form-Assembler and Remove-Variables, in a fashion similar to the combination of Sum and Squares in our earlier example. Since Form-Assembler and Remove-Variables each make a call on a more general function (Form-Assembler-From-Stmt and Remove-Variables-From-Stmt, respectively), our first step is to *generalize* Assemble, defining a new function Assemble-Stmt:

```
CONTEXT
   UNFOLDALL Assemble
   USING  Empty-Env InitialLabel
   USING RESTRICTED  Form-Assembler-From-Stmt
                     Remove-Variables-From-Stmt
   TRANSFORM
      GOAL Assemble(STMT,ADR) <=
                $$(&&Assemble-Stmt(STMT,Empty-Env(ADR),InitialLabel))
   END
END
```

This modifies Assemble and produces the definition

```
Assemble-Stmt :source-statement x environment x label -> list assembler
Assemble-Stmt(STMT,ENV,LAB)
            <= Form-Assembler-From-Stmt(
                        Remove-Variables-From-Stmt(STMT,ENV),LAB)
```

Observe that Empty-Env(ADR) has been generalized to ENV (a variable of type environment), and InitialLabel to LAB (a variable of type label).

Essentially, we have generalized the combination of Form-Assembler and Remove-Variables to the combination of Form-Assembler-From-Stmt and Re-

```
expression  <= expr(variable)  ++  application(operator , list expression)

source-statement <= assignment(variable , expression)
                 ++  while-loop(expression , source-statement)
                 ++  if-then-else(expression , source-statement ,
                                                  source-statement)
                 ++  block(list variable , source-statement)
                 ++  source-statement $ source-statement;

assembler <= load(address)  ++  store(address)  ++ jump(label)
          ++  jumpontrue(label) ++ apply(operator)  ++  nonop
          ++ labelled-assembler(label , assembler)
```

Fig. 3. Notation: To the left of the "<=" is the data type being defined, and to the right, separated by "++", are the cases of its definition, giving the constructor function for each. For example, *list* α <= nil ++ cons(α, *list* α). Also, "$" is being used as an infix constructor.

move-Variables-From-Stmt. Now we transform Assemble-Stmt to obtain a recursive definition not making use of either of these (similar to the transformation of SumSquares following its introduction). To follow some of the results of this, we must show more details of the specification:

```
    Remove-Variables-From-Stmt : source-statement x environment -> source-statement'
Remove-Variables-From-Stmt(assignment(V,EXPR),ENV)
            <= assignment'(Lookup(V,ENV),
                           Remove-Variables-From-Expr(EXPR,ENV))

(and so on for the other cases of source-statement)
```

```
        Form-Assembler-From-Stmt : source-statement' x label -> list assembler
Form-Assembler-From-Stmt(assignment'(ADR,EXPR'),LAB)
            <= Append(Form-Assembler-From-Expr(EXPR'), store(ADR))

(and so on for the other cases of source-statement')
```

To do the transformation, we give a GOAL for each case of *source-statement* that could be an argument to Assemble-Stmt (analogous to giving a GOAL for each of the cases of a list of numbers that could be an argument to SumSquares). For example, for the case assignment(V, EXPR) (where V is a variable, EXPR an expression) we give (within the appropriate CONTEXT):

```
GOAL Assemble-Stmt(assignment(V,EXPR),ENV,LAB)
            <= $$(&&Assemble-Expr(EXPR,ENV),V,ENV)
```

Note that this will lead to the introduction of another new function, Assemble-Expr, which in turn must be transformed.

Continuing this process leads to definitions of Assemble-Stmt, Assemble-Expr, and Assemble-List-Expr such that variable removal and formation of assembler code are now done simultaneously (avoiding the generation of the intermediate structure of source code with variables replaced by addresses), the behavior we sought from this optimization.

The equations specifying each of the new functions are these:

```
Assemble-Stmt(STMT,ENV,LAB)
          <= Form-Assembler-From-Stmt(
                              Remove-Variables-From-Stmt(STMT,ENV),LAB)
```

```
Assemble-Expr(EXPR,ENV)
          <= Form-Assembler-From-Expr(Remove-Variables-From-Expr(EXPR,ENV))
```

```
Assemble-List-Expr(EXPR-LIST,ENV)
               <= Form-Assembler-From-List-Expr(
                         Remove-Variables-From-List-Expr(EXPR-LIST,ENV))
```

Their transformed equations are

```
Assemble-Stmt(assignment(V,EXPR),ENV,LAB)
         <= Append(Assemble-Expr(EXPR,ENV), store(Lookup(V,ENV)))

 (and so on for the other cases of source-statement)
```

```
Assemble-Expr(expr(VAR),ENV) <= cons(load(Lookup(VAR,ENV)),nil)

Assemble-Expr(application(OPER,EXPR-LIST),ENV)
         <= Append(Assemble-List-Expr(EXPR-LIST,ENV),cons(apply(OPER),nil))
```

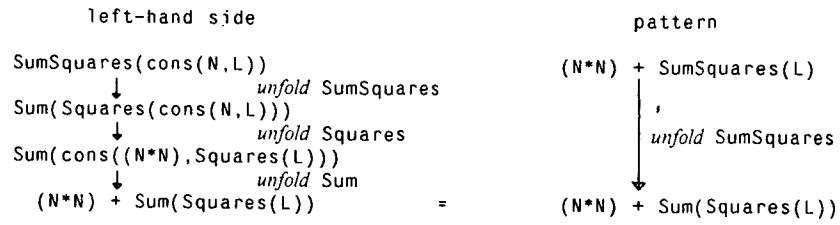
```
Assemble-List-Expr(nil,ENV) <= nil

Assemble-List-Expr(cons(EXPR,EXPR-LIST),ENV)
        <= Append(Assemble-Expr(EXPR,ENV),Assemble-List-Expr(EXPR-LIST,ENV))
```

4.3.4 *Linear Sequences of Manipulations.* We have stated all along that the bottom layer of our transformation hierarchy consists of the linear sequence of manipulations which actually make the changes to the recursion equations. Here we consider how the ZAP system performs the expansion from the pattern-directed transformation commands into such manipulation sequences.

The transformation GOALs consist of a left-hand side, the expression to be transformed, and a right-hand side, the pattern which expresses the answer sought. In the trivial case of no pattern being provided, the left-hand side is fully unfolded, and, provided the resulting expression is sufficiently trivial (we do not go into the precise meaning of this), it serves as the answer; the manipulation sequence is thus the sequence of unfold manipulations used to do the unfolding.

To understand what happens when a pattern is provided, first consider the simple case of no uses of "$$" or "&&". The system unfolds both the left-hand side and the pattern as far as possible. If these expand to identical expressions, then the pattern is a valid answer; the manipulation sequence is the sequence of "unfold" manipulations to unfold the left-hand side, followed by the inverse of the sequence of "unfold" manipulations used to unfold the pattern (i.e., a sequence of "fold" manipulations). An example is given in Figure 4.

The use of "$$" and "&&" within patterns is supported by replacing the equality test (between expanded left-hand side and expanded pattern) with a pattern match, where "$$" and "&&" are the pattern variables. The bindings formed in the match are used to instantiate the pattern, giving the exact pattern which, when expanded, equals the expanded left-hand side. See Figure 5. Here, the



Manipulation sequence:

	SumSquares(cons(N,L))
unfold SumSquares	Sum(Squares(cons(N,L))
unfold Squares	Sum(cons((N*N),Squares(L)))
unfold Sum	(N*N) + Sum(Squares(L))
fold SumSquares	(N*N) + SumSquares(L)

Figure 4

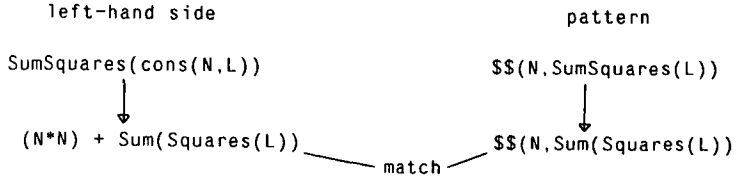


Figure 5

match binds $$ to λ a b . (a * a) + b, and instantiating the pattern gives (N * N) + SumSquares(L).

The final extension of this process is to make the match take into account declared properties of associativity and commutativity. If the matcher makes use of such properties, this corresponds to the application of the appropriate lemmas in the manipulation sequence. For example, suppose we declare "+" to be commutative and have the situation depicted in Figure 6. These are equal up to commutativity of "+"; so the manipulation sequence is as given in Figure 7.

Inspiration for inclusion of this last feature derives from Topor's matcher in his interactive verification system [26].

If we look at a portion of the compiler transformation, we see how much detail is captured by a GOAL step. Figure 8 gives the manipulation sequence ZAP generates to achieve the transformation expressed by

```
GOAL Assemble-List-Expr(cons(EXPR,EXPR-LIST),ENV) <=
          $$(Assemble-Expr(EXPR-LIST,ENV),Assemble-List-Expr(EXPR-LIST,ENV)
```

(part of the transformations of Section 4.3.3).

In truth, our system never actually extrudes the manipulation sequences justifying its activities, since we always construct and examine transformations at the higher (and more amenable) levels of our hierarchy.

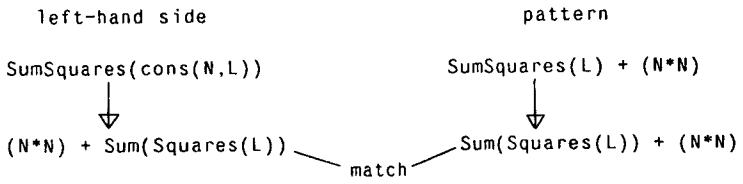


Figure 6

	SumSquares(cons(N,L))
unfold SumSquares	Sum(Squares(cons(N,L)))
unfold Squares	Sum(cons((N*N),Squares(L)))
unfold Sum	(N*N) + Sum(Squares(L))
apply commutativity of "+"	Sum(Squares(L)) + (N*N)
fold SumSquares	SumSquares(L) + (N*N)

Figure 7

```
Assemble-List-Expr(cons(EXPR,EXPR-LIST),ENV)
        |  unfold Assemble-List-Expr
        v
Form-Assembler-From-List-Expr(Remove-Variables-From
                    -List-Expr(cons(EXPR,EXPR-LIST),ENV))
        |  unfold Remove-Variables-From-List-Expr
        v
Append(Form-Assembler-From-Expr(Remove-Variables-From-Expr(EXPR,ENV)),
       Form-Assembler-From-List-Expr(Remove-Variables
                -From-List-Expr(EXPR-LIST,ENV)))
        |  fold Assemble-Expr
        v
Append(Assemble-Expr(EXPR,ENV),
       Form-Assembler-From-List-Expr(Remove-Variables
                    -From-List-Expr(EXPR-LIST,ENV)))
        |  fold Assemble-List-Expr
        v
Append(Assemble-Expr(EXPR,ENV),Assemble-List-Expr(EXPR-LIST,ENV))
```

Figure 8

4.4 Transformed Program

At the end of our entire transformation we emerge with a two-pass compiler program, still in recursion equations. The first pass converts source statements into assembler code and, simultaneously, constructs a mapping from labels to corresponding addresses. In the second pass the mapping is used to replace jumps to labels with jumps to explicit addresses, producing the final machine code.

When we convert to an imperative language, we might take advantage of side effects to combine these two passes into a single pass which destructively replaces (on the fly) forward referencing labels by addresses. The major change from

initial program (simple, modular, and multipass in nature) to final program *has* been accomplished by our transformation techniques.

5. REVIEW OF THE APPROACH

At this point we look over our approach, describe some enhancements to it (both incorporated and planned), and, finally, discuss the significant difficulties we foresee.

Figure 9 shows the hierarchical structure we use for transformation. Most of the effort the user must provide is in expanding transformation tactics into pattern-directed transformations, in the form of commands to ZAP. Some special devices have been incorporated into the system to support this operation.

5.1 Enhancements

The enhancements to ZAP in the form of special devices are designed to be used in commonly occurring, relatively straightforward transformations. They produce the left-hand sides of GOALs to transform a function, and simple recursive patterns to serve as entire patterns (or portions of patterns) of GOALs. They are called into action by the inclusion of keywords within the pattern-directed transformations.

5.1.1 *Producing the Left-Hand Sides of* GOALs. This special device generates the left-hand sides of GOALs to transform a function, using the data declarations of the type(s) of the argument(s) within the left-hand side to construct the cases. For example, to transform Sumsquares, we may write

GOAL SumSquares (CASESOF NUMLIST) <= ...

to direct the system to generate the cases for the arguments of SumSquares. To do this expansion, the system examines the data declaration for the type of the argument; here the type is *list number*, an instance of the parameterized type *list* α, with data definition

list α <= nil ++ cons(α, *list* α).

Each of the cases of the data definition's right-hand side suggests a case for the left-hand side of a GOAL,

```
SumSquares(nil);
SumSquares(cons(N, NUMLIST)).
```

5.1.2 *Producing Simple Recursive Patterns.* This special device generates simple recursive patterns to serve as entire patterns (or portions of patterns) of GOALs. As with the device to generate cases, the data declaration serves to provide the necessary information. For example, in transforming Sumsquares we have a goal with left-hand side

SumSquares(cons(N, NUMLIST)).

From the data definition of lists,

list α <= nil ++ cons(α, *list* α),

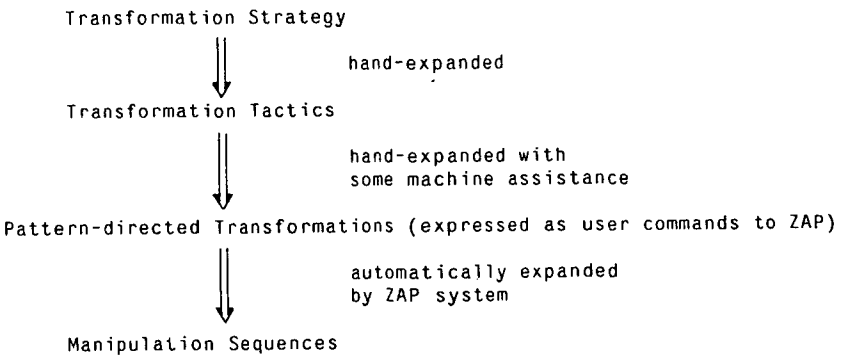


Fig. 9. Our transformation hierarchy.

the special device deduces that in dealing with a function call of the form Foo(cons(α, α-list)) a simple recursive call would be Foo(α-list). In our example this instantiates to

SumSquares(NUMLIST).

If the pattern of a goal contains the keyword AUTO (mnemonic for AUTOmatic), this is replaced by a simple pattern, $$ with arguments the recursive call found in the above manner, together with all the free variables of the left-hand side. So

GOAL SumSquares(cons(N, NUMLIST)) <= AUTO

is expanded to

GOAL SumSquares(cons(N, NUMLIST))
<= $$(N, NUMLIST, SumSquares(NUMLIST)).

free variables of left-hand side — recursive call

We may use this in our compiler example; for example,

GOAL Assemble-List-Expr(cons(EXPR, EXPR-LIST), ENV)
<= $$(Assemble-Expr(EXPR, ENV), AUTO).

AUTO expands to include a simple recursive call to Assemble-List-Expr, and the pattern produced is

$$(Assemble-Expr(EXPR, ENV),
$$(EXPR, EXPR-LIST, ENV, Assemble-List-Expr(EXPR-LIST))).

Note that, although this is more general than the pattern we would have typed in by hand (because it contains some unnecessary variables), the transformation has the same effect.

The CASESOF and AUTO devices may be used in conjunction; for example,

GOAL SumSquares(CASESOF NUMLIST) <= AUTO

expands to

```
GOAL SumSquares(nil) <= $$()
GOAL SumSquares(cons(N, NUMLIST))
     <= $$(N, NUMLIST, SumSquares(NUMLIST)).
```

5.1.3 *Motivation for Nature of Enhancements.* Our motivation for the inclusion and method of activation of these devices comes from the observation that often, during expansion of a tactic into pattern-directed transformations, very simple cases and patterns are required. These we sought to automate, so that the user would be freed to concentrate upon the more complex portions where his intelligence is needed. We still require the user to activate them (rather than always attempting to apply them), reasoning that this gives the user the appropriate degree of control. Furthermore, it is easy for the user to override them should they fail.

We would like to see further development of the system fit into this framework. For example, we might automate expansion of our simple strategy so that, given a program to transform, the system suggests a sequence of tactics to implement that strategy; the user could then follow that sequence, perhaps modifying it slightly where his insight suggested necessary divergences.

5.2 Other Transformations

Two other sizable transformations (in the same class of problems as the compiler example) that we have performed with the aid of our system are these:

(1) *The Telegram Problem* (from [16]). This problem involves decoding an incoming stream of characters representing telegrams and amassing statistics about the telegrams. This problem served as the first major example upon which we developed our approach and system. Reference [14] discusses the transformation. The scale of the problem (judged in terms of number of recursion equations defining the initial program) was about half the size of the compiler example (some 30 equations as compared to 60).

(2) *A Text Formatter.* We adopted the formatter described in [17, chap. 7]. This program takes as input text interspersed with commands to direct the layout of that text. Typical formatting operations, such as filling and justifying lines to align the right margin, centering text between margins, providing page numbers and titles, etc., are supported. We wrote a recursion equation program to perform all their formatting operations but concentrated on structuring the program in as straightforward a manner as possible. Our resulting program was really very different from, and much simpler than, Kernighan and Plauger's. See [12] for details of this specification. In terms of scale this was much larger than the compiler example (over 200 equations). In terms of difficulty of carrying out the transformation, the effort required had grown more than linearly with the number of equations. In retrospect we see that the difficulties were already present in the transformation of the compiler; however, its smaller scale made them relatively innocuous. What these difficulties were we consider next. In spite of the difficulties, we were successful in completing the transformation of this sizable program. We believe this to be the largest machine-assured transformation carried out to date.

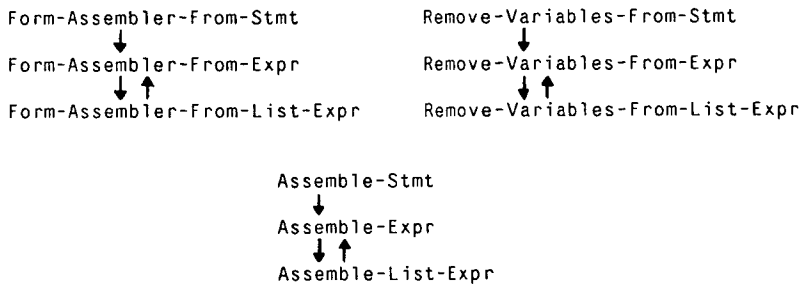


Figure 10

5.3 Difficulties

We have already pointed out that the expansion from tactics to pattern-directed transformations consumes most of our effort. Disturbingly, the effort required seems to be growing faster than the scale of the programs we are transforming. Of course, programming effort in following conventional development methods also increases nonlinearly with the size of the program.

We identify several sources of difficulty encountered during transformation in the following subsections.

5.3.1 *Weakness of Patterns.* Although we may include within our patterns variables ($$ and &&) to abstract away some of the detail, as the recursion equations get larger and more complex (as is the case during longer transformations) we still find ourselves having to construct larger and more complex patterns. To overcome this, we might

(1) extend the enhancements to provide more system-generated portions to be included in patterns or
(2) make our patterns more general so as to be able to abstract away more of the detail when describing the desired goal of a transformation. We have only begun to consider how this might best be done.

5.3.2 *Repeated Structure of Transformation.* Examining a portion of the compiler transformation (Section 4.3.3), we can see a similarity between the structure of the initial recursion equations and the structure we transform into; see Figure 10.

In writing pattern-directed transformations we are forced to repeat this structure by hand—an annoying situation that tends to occur repeatedly within larger transformations. Clearly, there is need for some mechanism to abstract out the algorithmic structure which we can then use to assist in generation of transformation commands. Such a mechanism should integrate well with the CASESOF and AUTO devices.

5.3.3 *Sensitivity of Tactics to the Structure of Functions.* At the tactics level our hope was that the expansion of tactics would not depend crucially upon the entire structure of the recursive functions. In examining the compiler transformation we see that the application of the tactic to combine Assemble and Remove-Labels in fact did depend crucially on the way in which these functions

called other lower level functions. In fact, our very first action was to perform a simple generalization: the application of a tactic not predicted at the strategy level. We might consider this an omission due to naïveté when expanding strategy into tactics or, alternatively, conclude that in expanding tactics we might generate further applications of tactics. Clearly, the algorithmic structure is important in our transformation process. As we increase the scale and complexity of programs to be transformed, the algorithmic structure becomes deeper and more complex, and we observe this phenomenon more and more.

6. CONCLUSIONS

We have presented an approach to applying a transformation method to programs of increasing scale. Our experience in performing several sizable transformations has demonstrated success in this regard. We have identified areas where our techniques require improvement and have made some suggestions as to how this might be done. In addition to continuing investigations into performing larger and more complex transformations, we see wide scope for investigating how transformation might assist in program modification and maintenance. To complete the viability of our approach, we recognize the need for further research into the conversion of applicative programs into imperative programs making use of destructive operations on appropriate data structures. Further detail about the work described here may be found in [13].

ACKNOWLEDGMENTS

We would like to thank all our past colleagues at the Department of Artificial Intelligence, University of Edinburgh, for their help, and in particular Rod Burstall and John Darlington for providing the original inspiration and continuing encouragement and assistance throughout. The referees provided valuable and penetrating comments and suggestions.

REFERENCES

1. BALZER, R., GOLDMAN, N., AND WILE, D. On the transformational implementation approach to programming. In Proc. 2d Int. Conf. Software Engineering, San Francisco, Calif., Oct. 1976, pp. 337–344.
2. BAUER, F.L., PARTSCH, H., PEPPER, P., AND WÖSSNER, H. Techniques for program development. In *Infotech State of the Art Report: Software Engineering Techniques.* Infotech Information Ltd., Maidenhead, Berkshire, England, 1977, pp. 25–50.
3. BAUER, F.L., PARTSCH, H., PEPPER, P., AND WÖSSNER, H. Notes on the project CIP: Outline of a transformation system. Tech. Rep. TUM-INFO-7729, Inst. für Informatik, Technische Univ. München, 1977.
4. BOYLE, J.M. Program adaption and program transformation. In *Practice in Software Adaption and Maintenance: Proceedings, Workshop on Software Adaption and Maintenance, Berlin.* Elsevier North-Holland, New York, 1979, pp. 3–20.
5. BURSTALL, R.M. Design considerations for a functional programming language. In *The Software Revolution: Proc. Infotech State of the Art Conference, Copenhagen.* Pergamon Press, Elmsford, N.Y., 1977, pp. 45–57.
6. BURSTALL, R.M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44–67.
7. BURSTALL, R.M., MACQUEEN, D.B., AND SANNELLA, D.T. HOPE: An experimental applicative language. In Proc. 1980 LISP Conf., Stanford, Calif., 1980, pp. 136–143.

8. DARLINGTON, J. A synthesis of several sorting algorithms. *Acta Inf. 11*, 1 (Dec. 1978), 1–30.
9. DARLINGTON, J. Program transformation and synthesis: Present capabilities. Tech. Rep. DAI 43, Dep. Artificial Intelligence, Univ. Edinburgh, Edinburgh, Scotland, 1977; also appeared in *Artif. Intell. 16*, 1 (March 1981), 1–46.
10. DARLINGTON, J. A Semantic Approach to Automatic Program Improvement. Ph.D. dissertation, Dep. Artificial Intelligence, Univ. Edinburgh, Edinburgh, Scotland, 1972.
11. DARLINGTON, J., AND BURSTALL, R.M. A system which automatically improves programs. *Acta Inf. 6*, 1 (March 1976), 41–60.
12. FEATHER, M.S. Program specification applied to a text-formatter. Available from author; submitted to *IEEE Trans. Softw. Eng.*
13. FEATHER, M.S. A System for Developing Programs by Transformation. Ph.D. dissertation, Dep. Artificial Intelligence, Univ. Edinburgh, Edinburgh, Scotland, 1979.
14. FEATHER, M.S. Program transformation applied to the telegram problem. In Proc. 3d Int. Symp. Programming, Paris, 1978, pp. 173–186.
15. FRIEDMAN, D.P., AND WISE, D.S. Functional combination. *Comput. Lang. 3*, 1 (Feb. 1978), 31–35.
16. HENDERSON, P., AND SNOWDON, R. An experiment in structured programming. *BIT 12*, 1 (1972), 38–53.
17. KERNIGHAN, B.W., AND PLAUGER, P.J. *Software Tools.* Addison-Wesley, Reading, Mass., 1976.
18. KOTT, L. About a transformation system: A theoretical study. In Proc. 3d Int. Symp. Programming, Paris, 1978, pp. 232–247.
19. LEE, S., DE ROEVER, W.P., AND GERHART, S.L. The evolution of list-copying algorithms and the need for structured program verification. In Conf. Rec., 6th Ann. ACM Symp. Principles of Programming Languages, San Antonio, Tex., Jan. 29–31, 1979, pp. 53–67.
20. MANNA, Z., AND WALDINGER, R. Synthesis: Dreams ⇒ programs. *IEEE Trans. Softw. Eng. SE-5*, 4 (1979), 294–328.
21. MARTELLI, A. Program development through successive transformations: An application to list processing. In Proc. 3d Int. Symp. Programming, Paris, 1978, pp. 381–394.
22. PARTSCH, H., AND PEPPER, P. Program transformations on different levels of programming. Tech. Rep. TUM-INFO-7715, Institut für Informatik, Technische Univ. München, 1977.
23. PETTOROSSI, A. Improving memory utilization in transforming recursive programs. In *Proceedings, 7th Symposium on Mathematical Foundations of Computer Science, Zakopane, Poland.* Springer-Verlag, New York, 1978, pp. 416–425.
24. PETTOROSSI, A. Transformation of programs and use of "tupling strategy." Presented at Informatica 77, Bled, Yugoslavia, 1977.
25. SCHWARZ, J. Verifying the safe use of destructive operations in applicative programs. In Proc. 3d Int. Symp. Programming, Paris, 1978, pp. 395–411.
26. TOPOR, R.W. Interactive Program Verification Using Virtual Programs. Ph.D. dissertation, Dep. Artificial Intelligence, Univ. Edinburgh, Edinburgh, Scotland, 1975.
27. WEGBREIT, B. Goal-directed program transformation. *IEEE Trans. Softw. Eng. SE-2*, 2 (1976), 69–80.

Received August 1979; revised September 1980 and April 1981; accepted July 1981